UiT The Arctic University of Norway

The Faculty of Science and Technology
Department of Computer Science

A Data Gathering System for the Arctic Tundra

Jørgen Aleksander Larsen

Master's thesis in Computer Science INF-3981 – June 2024

Supervisors

Main supervisor:	John Markus Bjørndalen	UiT The Arctic University of Norway, Faculty of Science and Technology, Department of Computer Science

To my parents.

“Debugging is like being a detective in a crime movie where you’re also the murderer.”
–Filipe Fortes

“A good programmer is someone who always look both ways before crossing a one-way street.”
–Doug Linder

Abstract

Climate change has emerged as an important topic over the past decade, and one of the areas most susceptible to change is the Arctic Tundra. Monitoring the environment features a variety of challenges; it's remote location, manual monitoring equipment and required permission to depart on expeditions. A solution to this is the use of a wireless sensor network to allow more automatic gathering of data. Many algorithms to increase the life span of nodes have been proposed over the years, such as LEACH and PEGASIS. However, these make assumptions that does not fit the Arctic Tundra.

This thesis proposes a system design which minimizes message propagation as it aims to overcome the networking challenges, while also limiting energy consumption. The system consists of two types of nodes; normal sensor nodes, and relay nodes which communicates with a base station. Relay nodes will inform others of its presence and set paths are created through the system so all nodes can propagate their data. Some of the challenges with simulating such a system is explored, and it is implemented on top of an event-based simulator.

Experiments are run to evaluate the energy consumption of the system using a combined energy model from LEACH and ESDS, as well as the scalability of the simulator. The results showcase that most of the energy is expended by being awake, and a very small part is due to sending messages. Additionally, it means that the proposed system is mainly viable for smaller networks with sparsely placed nodes. No real conclusion can be made about the model scalability results, other than increasing the simulated time will increase the simulation run-time. As an example, a year can be simulated by running for approximately 30 minutes.

Finally, due to messages being such a small part of the energy consumption, this opens up for many interesting approaches. The main one presented being mesh networks, as this allows algorithms such as LEACH and PEGASIS to overcome the networking assumptions as the problems of routing is handled in an underlying network layer.

Acknowledgements

I would like to thank my advisor John Markus Bjørndalen for providing support and encouragement during the rough patches of this thesis. Also, thank you for the many long discussions about the topic and chit-chats during our meetings.

Thank you to the friends I made along the way for all the collaborations and fun throughout the years.

Thank you to my family for their endless support during this period.

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

Climate change is an increasingly discussed topic over the past decade as a focus on sustainability on earth becomes a bigger focus. One of the areas most susceptible to change is the Arctic Tundra. The average temperature has risen significantly, causing ice to melt, and ground and soil to spring forward. This allows new invasive species to infiltrate the ecosystem and push other species to new areas. The changing weather causes the snow to change in consistency, affecting some of the species. Therefore, it is important to monitor these changes so we can further research the effects they cause.

Monitoring the Arctic Tundra features a variety of challenges. First is its remote location, often requiring long expeditions spanning several days of travel. Secondly, much of the monitoring equipment used requires field personnel to manage updates and collect data. This is a result of equipment often being conservative with its hardware and not having proper communication capabilities[1]. Finally, departing on these expeditions requires permission which might take several years to acquire[2, 3]. The Distributed Arctic Observatory[4] (DAO) is a project aimed at solving such problems with Climate-ecological Observatory for Arctic Tundra[5] (COAT) as a use case.

A Wireless Sensor Network (WSN) is an important paradigm in use to monitor environments and gather necessary data. This is a collection of sensor units which communicate and propagate data between themselves over a network to a base station. These networks can consist of hundreds of nodes.

One of the challenges WSNs face is power consumption. This is also one of the major problems sensors face in the Arctic Tundra. The location makes nodes unable to include a constant power source, instead relying on battery power. Replacing these batteries require monumental effort.

Algorithms such as LEACH[6] (Low-Energy Adaptive Clustering Hierarchy) aims to reduce the power consumption and increase the lifetime of sensor nodes. However, one of the problems with many of these algorithms is their assumption of all nodes being in range of each other and a base station. This is often not the case in an environment such as the Arctic Tundra, where networking capabilities are limited. This shows the need for a system that can tackle these problems.

One of the biggest difficulties with the Arctic Tundra is how to test proposed solutions to these problems. In other words, how to test systems aimed at solving them. The ideal way to test a system is to build and deploy it, as this gives the most accurate data. However, this is often not feasible due to the requirements of acquiring hardware, implementing the system on it, and travelling to deploy the nodes.

Instead, cheap but similar hardware can be acquired to create simple prototypes on, such as a Raspberry Pi. The other alternative is to create a pure software prototype on a normal computer. This software prototype can either be creating local servers for a simulation using an actual network, or it can be implemented in a simulator. This gives the options of either running a real-time simulation, or a simulated time simulation. The difference is that a real-time simulation needs to wait for downtime, whereas a simulated time one can skip over it. One of the problems with running simulations is that many assumptions are made about the environment and problems are often abstracted away. This is a trade-off that needs to be kept in mind.

Using a simulator also has its own set of challenges, especially when it comes to choosing one. Many options exist today such as NS3[7], OMNeT++[8] or Simpy[9]. These are all simulators with different abstraction levels, some are more fine-grained and complex than others. What these all have in common is them being discrete event simulators. Simpy is more generic, whereas NS3 and OMNeT++ are more fine-grained packet-level simulators with a focus on networking. One of the problems with these simulators is their complexity, making it hard to create proper environments matching that of the Arctic Tundra.

1.1 Thesis Statement

The main reason LEACH cannot be directly utilized in the arctic is due to how the algorithm ignores the networking challenges of the Arctic. However, it is interesting because it is aimed at lowering the energy consumption of nodes. To further help understand these problems, the thesis is focused on the following problem statement:

> *This thesis aims to explore the challenges of propagating sensor data throughout a sensor network in an environment like the Arctic Tundra while keeping in mind how the energy consumption is affected.*

1.2 Scope and Limitations

The scope of this thesis is focused around designing a system to propagate data between nodes. No actual sensor data is being simulated, instead the typical size of a message is used in relevant calculations. This means that the sensing part of nodes is abstracted away, and any implementation instead uses a test message. More of the system specific scope will be presented in chapter 5 where system design is covered.

When first designing the system, a bigger scope was set where data is being sent from multiple clusters to a base station. However, due to time constraints, the final scope ended up being focused on the internals of a single cluster.

This thesis also presents some of the challenges of simulating a system in the Arctic. While it is not the main contribution of this thesis, the work was significant enough to designate a chapter for it, this being chapter 4.

1.3 Contributions

The contributions of this thesis can be seen in two parts, a data gathering system and the simulator. It can be summarized as follows:

- Propose a design for a data gathering system in the Arctic Tundra.
- An event-based simulator to study the proposed system.
- An experiment to study the energy consumption of the data gathering system.

- An experiment to evaluate the scalability of the simulator.
- A discussion around the results of the experiments, as well as the overall design choices made for both the system and the simulation.

1.4 Outline

The thesis is organized in the following way:

Chapter 1 - Introduction: Has just introduced the motivation of the project, along with a problem statement and contributions.

Chapter 2 - Background: Presents some background knowledge which is useful to know for this thesis.

Chapter 3 - Related work: Presents work in areas related to this thesis.

Chapter 4 - Simulator: Introduces some of the challenges around prototyping, and presents the design and implementation of the simulator.

Chapter 5 - Data Gathering System: Presents the architecture, design and implementation of the proposed system of this thesis.

Chapter 6 - Evaluation: Presents the methodology, goal and results of the experiments run.

Chapter 7 - Discussion: Discusses various topics around the results, the proposed system, it's weaknesses, and other interesting areas.

Chapter 8 - Future Work: Presents some of the future work which can be done, both academically and for the implementation.

Chapter 9 - Conclusion: Concludes the thesis.

2 Background

2.1 Wireless Sensor Network

A wireless sensor network is a collection of smaller sensor nodes which can measure and gather information about the surrounding environment. These nodes usually have very limited hardware capabilities in the form of available memory and battery lifetime. Each node transfers data to a base station, which can be an access point or a form of handheld device[10]. The transfer can happen either through direct communication or a mesh network.

There are two kinds of WSNs. The first one is an unstructured network. This is a dense collection of nodes which are mostly placed randomly in the field. The second one is a structured network. This is a more sparse collection of nodes, where each node is placed in an intentional and pre-planned location[10].

2.2 Network Routing Types

There are two different ways of allowing nodes to communicate with each other. The first one is if all nodes are in range of each other and can be referred to as a single-hop routing. Nodes can communicate directly over cable or wirelessly over radio.

The second one is referred to as multi-hop routing and is utilized if nodes cannot communicate directly. This is done by having intermediate nodes act as relays to forward messages to other intermediate nodes or the target node. One of the most common schemes is to minimize the hop-count of messages, but many other factors can also be used to choose the path taken, such as energy levels or signal strength[11].

2.3 Mesh Networks

A mesh network is an example of a system which utilizes multi-hop routing where nodes forwards traffic due to limited range. The system generally consists of mesh routers and mesh clients. Each node in the system is both a host and a router. A wireless mesh network operates dynamically and automatic, meaning it organizes and configures itself to maintain connectivity between nodes[12].

2.4 Event-based Programming

In computer science the word event is very vague as its meaning is dependent on the context. An event is often defined as an occurrence of some kind, and this occurrence can take many forms. For an operating system, this could be movement or the click of a mouse. Applications on top of the OS can then respond to these events however they like. Design patterns such as "subject-observer" is based on events, and is used when an observer wants to know about the events of a subject, much like in a publish-subscribe system[13].

Tarkoma has a focus on publish-subscribe systems, but defines an event as a discrete state transition[14]. Much like Faison, Tarkoma also connects conditions, events and notifications together. If something happens which matches a condition, and subsequently a notification is created, then that is an event[13].

In programming languages, events often has its own definition as well. According to Faison, events are often associated with delegates, event variables and handlers in .NET. Whereas in Java, it is associated with event sources, event objects and handlers[13]. These terms closely related to event-based programming. Much like is described with the subject-observer pattern, it often means some form of loop is polling for events in a queue, and sources will create these events. A handler is used to react appropriately[15].

2.5 Asyncio

Asyncio is a library in Python used for asynchronous programming. It is specifically used to write concurrent code. It is important to note that it does not use threads for parallel programming nor multiprocessing. Instead, it operates inside a single process. It uses coroutines which can be scheduled concurrently.[16].

The concept it utilizes is asynchronous programming. The important part of this is that a routine can pause while waiting for the result of some action it has taken. During this pause, other routines in the schedule can be run. This is what allows for concurrency to take place, as multiple actions can overlap through this paradigm[16].

Asyncio implements this concept of concurrency by utilizing generators in Python and adds the keyword combination of async/await. A function can be defined as async and the await action allows it to yield control of the single thread to a different coroutine[16].

3 Related Work

3.1 LEACH

Low-Energy Adaptive Clustering Hierarchy[6] (LEACH) is a protocol aiming to decrease the total energy consumption of wireless sensor networks and increase the lifetime of the nodes in the system. The algorithms make some assumptions about hardware such as using a first-order radio model, and each node being able to reach a base station.

The paper proposes a clustering algorithm consisting of a few different phases; advertisement, cluster set-up, schedule creation and data transmission. Nodes will randomly elect themselves as cluster heads and send out election messages. Nodes will accept the cluster with the strongest radio signal. Each cluster will then send out a schedule for each cluster member, giving them a time slot to send their data. Each node will send their data and go to sleep. These phases define a round in the system.

Many variations of this algorithm exist, and it is a widely researched area of energy conservation in wireless sensor networks. Some examples of this is LEATCH[17], which defines a two-level cluster hierarchy. This means that there is a super cluster which contains many smaller cluster heads from the original LEACH protocol. Another example is LEDCHE-WSN[18], this variation assumes there are many different clusters, and each cluster has a set sink node which can communicate with the base station.

The system in this thesis differs greatly from LEACH and LEATCH in the sense that this thesis breaks the assumption of all nodes being able to communicate with the base station. However, a similarity can be seen with LEDCHE-WSN which utilizes a sink node. This serves the same purpose as the relay nodes proposed in chapter 5 by connecting the base station with the cluster.

3.2 PEGASIS

Power-Efficient Gathering in Sensor Information Systems[19] draws inspiration from LEACH to further enhance the energy efficiency of a WSN. It builds upon the same assumption as LEACH where each node can reach a base station or any other node. It works by forming a chain between neighboring nodes and having data transmit between the links. A random node in this chain is responsible for sending the data in each round. This is important to ensure that nodes dies at random locations. When a node dies, the chain is rebuilt by simply bypassing it.

PEGASIS is similar to the system in this thesis in the sense that chains are formed between nodes to forward data. But again, the assumption of reaching all nodes is not the same. Additionally, PEGASIS forms chains where the destination changes, whereas the system in this thesis forms chains in a multi-hop manner for a specific destination. This being a relay node.

3.3 TEEN

Threshold sensitive Energy Efficient sensor Network[20] (TEEN) is a protocol which targets reactive networks. LEACH and PEGASIS can be described as proactive networks, meaning they constantly monitor the environment at set intervals, whereas TEEN reacts to changes in the environment. TEEN also operates in a hierarchical cluster formation, where data aggregates up the cluster-levels until it reaches the base station.

The protocol defines two values, a hard threshold and a soft threshold. The hard threshold is set as the condition a sensed attribute has to reach before the transmitter is turned on and data is reported. The soft threshold is a used as a minimum difference that the previous sensed value and the new value has to reach before transmitting data. The first threshold eliminates unwanted data, and the second threshold limits reporting when there only are small changes in the environment.

The most important difference between TEEN and this thesis is TEEN being a reactive network and this thesis being a proactive network. There is little similarity, but TEEN showcases an important alternative to research on wireless sensor networks and the various applications it can be used for.

3.4 Clock Synchronization between Observational Units in the Arctic Tundra

"Clock Synchronization between Observational Units in the Arctic Tundra"[21] is a master's degree written at the University of Tromsø - The Arctic University of Norway, by Sigurd Karlstad. As the title alludes to, this degree focuses on exploring clock synchronization between the nodes. This topic is completely abstracted away in this thesis, but is a very important problem to cover in the Arctic Tundra as nodes are required to operate in tandem. The thesis presents work collaborated on between two students, and the overall work done showcases a larger system aimed towards the Arctic Tundra. This means it also includes a system design and how it can be implemented using servers and processes, compared to this thesis using an event-based queue.

3.5 Simulators

Many of the simulators mentioned in the introduction can also be seen as related work, as it is a relevant part to this thesis. However, these simulators are briefly described and related to this thesis in chapter 4.2 and are therefore not described here.

4 Simulator

This chapter describes some of the challenges faced when prototyping a system, and briefly presents some of the existing simulator options. It also presents the design and implementation of the simulator.

4.1 Prototyping

As mentioned in the introduction, there are many different ways to create a prototype of a system. Each way comes with its own set of pros and cons, as well as challenges and complexities. This thesis wants to explore the energy consumption of a system. Therefore, a physical prototype running on proper hardware is out of the question due to both costs and time constraints. Emulating the system using server processes is less expensive but still time-consuming, meaning testing 24 hours of run-time for the system requires 24 real hours. This limits the possible length of time we can run the system.

To solve this, an event-based simulator can be used. It is a good fit for a distributed monitoring system as they often report data at set intervals and powers off to save energy. This can be modelled as events and used to manipulate the simulated run-time such that if nothing happens it jumps to the next event, which may be hours in the future.

One of the trade-offs for using such a simulator is the ease at which kind of data can be obtained. Obtaining energy data is much simpler as it has to be modelled, but running benchmarks to obtain information about CPU and memory requirements is much harder, but not necessarily impossible. Simulations like this also requires implementing various abstractions such as the network. This may cause information leakage between abstraction layers which doesn't exist in the real world. It is important to be aware of these and understand how they can affect the results.

4.2 Existing Simulators

There exists many different discrete event simulators. As mentioned previously, NS3[7], OMNeT++[8] and Simpy[9] are all possible simulator options. Both OMNeT++ and NS3 are written in C++ and are defined as packet level simulators. Due to the combination of these being a lower abstraction level than needed, as well as being in an unfamiliar language, they were deemed too complex to utilize for this thesis. Simpy was also briefly explored, but due to its generic nature it was hard to relate the various types of resources and concepts to create abstractions such as networks and broadcast pipes.

Researchers at the Cyber Physical Lab at UiT has also worked closely on problems with simulating systems for the Arctic Tundra and has published a paper about a framework called "Extensible Simulator for Distributed Systems"[22] (ESDS). The progress of this tool was learned about towards the middle of work with this thesis, and was explored as an option to utilize for the system described in the next chapter.

ESDS is highly relevant to this thesis and was close to being a good fit. However, some of the problems encountered was it seemingly being strict on a node only having a single role such as sender or receiver instead of both. Additionally, the system in the next chapter utilizes multi-hop routing. Each of the links would then have to be defined in ESDS' configuration files.

Due to the problems above, this thesis implements the simulation from scratch instead of using any of the presented simulators.

4.3 Simulator Design

As alluded to from before, an event-based simulator is implemented due to being a good fit for the Arctic Tundra. An event queue is implemented and each event defined manually instead of relying on an existing framework. Parts of the simulation also take inspiration from ESDS. Firstly is the approach where a node's runtime behavior is defined in an execute function and evaluated during runtime to fill the queue. A similar paradigm is utilized here, but instead of evaluating this function at runtime, it is instead executed to fill the event queue before starting the simulator. This means that when the queue is empty, the simulation is complete without the need for much of the synchronization logic ESDS utilizes.

Figure 4.1 showcases a simple flowchart of the simulator. The program process starts, and the event queue is filled. Events are fetched, and if one exists, the simulated time is updated and the event is processed. Once no more events are in the queue, the simulation is complete.

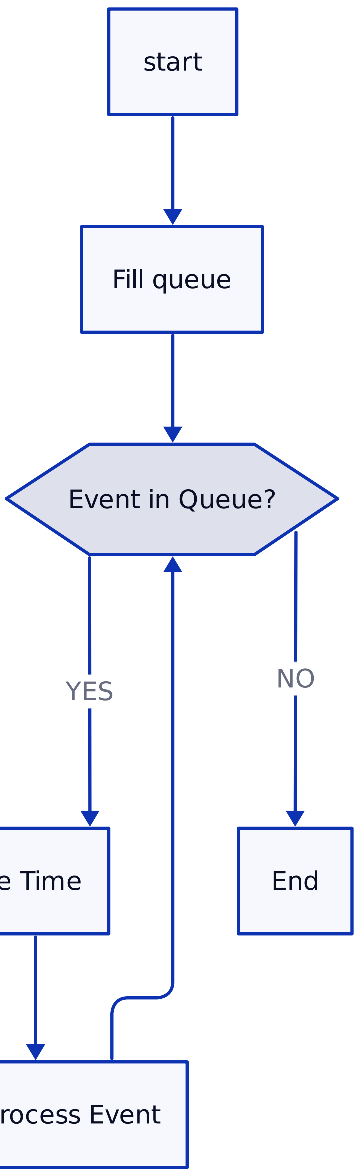

Figure 4.1: Simulator Flow

4.4 Programming Language

The simulator is implemented using Python. The reason for this is Python being a familiar language used a lot during previous years at UiT. It is not a compiled language and is interpreted at runtime. This makes it a good choice for implementing prototypes, as testing various approaches is a simple task without much overhead. Its simplicity has also caused its adoption by scientists and statisticians, making it a very good choice for this thesis.

4.5 Event Implementation

The simulator implements a generic event class which functions as a base for all the events to inherit. This includes a timestamp, the node which created the event, a content attribute, and a type field which simply contains the type of event as a string. The type attribute is mainly used to differentiate between events. It also features a generic print function which changes based on the type.

The main reason for defining events as classes is to have each type include an execute function where the functionality of the event is implemented. This makes it easy to add new types of events without having to define and expand a large event handler function to perform this task. Instead, when fetching events, this execute function is simply called without any other logic required.

4.6 Event Queue Implementation

The most important part of the simulator is to ensure that events are handled in the correct order. As this is a time-based simulation, the natural choice is to sort events based on a timestamp. This is done by utilizing Asyncio's Priority Queue. This takes a tuple as an input and always tries to sort on the first element. This gives us a tuple in the form of (timestamp, event object).

However, if two events have the same timestamp it will iterate to the next element, a class object in this case. This requires implementing a "less-than" magic method for the generic event class as otherwise Python will throw an exception. This allows for adding a priority within events if needed, however this is solved in other ways as will be explained in the next chapter. Therefore, the current implementation simply returns a boolean to stop the exception being thrown.

4.7 Configuration

The simulator itself has its configuration defined in a file named "config.py". This file contains multiple flags used to toggle various print options, as well as logging to files. Many of these are simply for debugging and to help isolate problems by not flooding all types of prints in the terminal or files. However, there are a few other variables related to the simulator such as network delay, node up-time and simulation time in hours. These are highly coupled to the implementation of the system itself and will therefore be mentioned in the next chapter. It also contains some other variables such as energy models and message size, which is also related to the system itself.

5 Data Gathering System

This chapter begins by introducing how LEACH was used as inspiration, before presenting the assumptions used when designing the system. The architecture part will define the node types, as well as hardware, topology, routing and its phases. The design covers the energy model and messages. Finally, the implementation covers the network grid and events, along with implementation specific details from the previous sections.

5.1 LEACH as Inspiration

This thesis originally set out to explore the challenges of using the LEACH algorithm in the Arctic Tundra. However, as mentioned in the introduction, the network assumptions made by LEACH[6] ignores some of the crucial challenges faced in the area. The most important one being the assumption that all nodes has a direct connection to all the other ones and a base station. In the Tundra nodes are not expected to have this connection due to limited network capabilities, which plays a crucial part in the algorithm.

Some work was done on a simple emulation of LEACH using local servers and network. However, the aforementioned challenges quickly appeared, causing a reevaluation of the approach. This resulted in designing a new system from scratch to get around these problems. Additionally, problems with synchronization quickly became apparent due to how LEACH operates in rounds.

An example of this is when a node wakes up in LEACH, it checks if it becomes a cluster head. If yes, then it broadcasts an election message, and all the other nodes needs to be in the start of the phase where they are waiting for this message. It then completes all its phases as described in the related work chapter and starts a new round by checking if it should be a cluster head again.

5.2 Assumptions when Designing the System

With the problems prompted by LEACH, a set of assumptions needs to be specified as a frame around the system. First, all nodes will not be in range of each other or a base station, but should at least see one other node. Secondly, a message should be able to reach any node through forwarding. Third, all nodes are synchronized and operate in tandem. Finally, nodes are assumed to be heterogeneous, meaning that they don't necessarily need to have the exact same hardware. This is important for the next section.

As alluded to in chapter 1.2 (scope and limitations), a complete data gathering system is assumed to consist of a base station which gathers all the data in a centralized storage and can send this data to the rest of the world by having proper internet access. Spread throughout a section of the Arctic Tundra are multiple clusters which gain access to the base station by utilizing some form of hardware with better communication capabilities, or multi-hop communication through other clusters.

This thesis will focus on the internals of a single cluster and some of the challenges this faces. The system design aims to minimize the amount of messages required. This is due to the networking challenges, as well as wanting lower energy consumption.

5.3 System Architecture

A cluster is assumed to include an area of around 500x500 meters where various types of nodes can be placed. This system defines two different types, which is why nodes are referred to as heterogeneous.

- A normal *node* is an observation unit whose main goal is to gather sensor data from the environment and store it.

- A *relay node* can also function as an observation unit and be equipped with sensors, but it is not a requirement. The important distinction is that the relay node is assumed to be equipped with more communication hardware to help reach the base station.

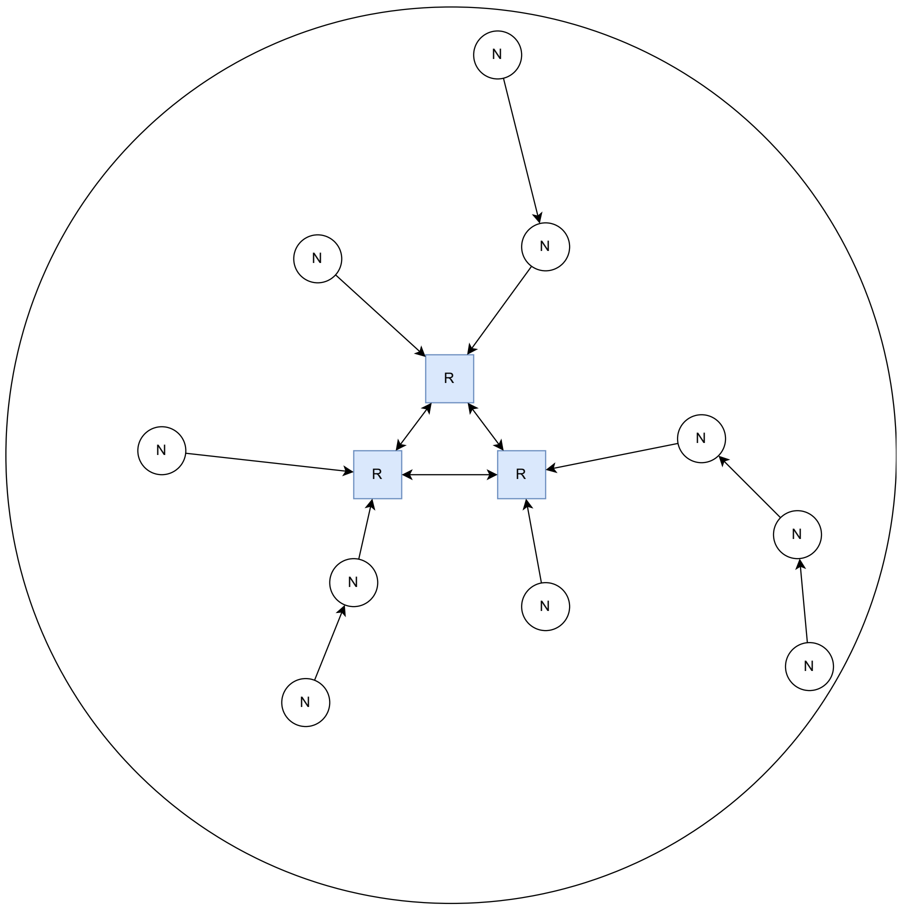

Figure 5.1: System Architecture

Figure 5.1 shows a simple example of how a cluster can be set up. It features three central *relay* nodes which are responsible for gathering all the data. As well as a set of normal nodes which reads the sensors and forwards this data through intermediate nodes until it reaches a *relay*.

5.3.1 Hardware Assumptions

Another important assumption is what kind of hardware the nodes utilizes, specifically for communication and networking. Due to the assumption of limited networking capabilities, inspiration is taken from LEACH[6] to utilize a first order radio-model. This means that communication happens over radio frequencies. Broadcasting is assumed to happen on a well known open frequency for all nodes to listen to. When sending messages to specific nodes, CSMA (carrier-sense multiple access) MAC can be used much like LEACH, for example based on the target node's id. However, this is abstracted away in the simulation, as the problem of efficient channel management is not a focus of this thesis.

5.3.2 Topology

Due to constrained networking capabilities and limited radio range, a form of multi-hop routing is required for nodes to be able to communicate. Figure 5.2 is an example of how the network topology looks like for nodes. Something like a partial mesh-like topology structure where messages can be forwarded through other nodes can solve this problem. This can also be seen in the proposed system architecture in figure 5.1 through the arrows connecting the nodes. It is important to note that the system does not utilize a mesh network, which will be explained later in the routing section.

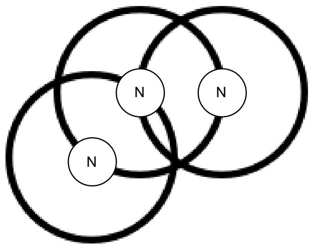

Figure 5.2: Network Topology

5.3.3 Node Isolation

To support the goal of minimizing the amount of messages being sent, the system does not utilize pings and health-checks between nodes. This means that each node cannot build a local view of the network, and has limited knowledge of which nodes are in range and no knowledge of their current status such as battery. As a result, no routing algorithm requiring dynamic state information about other nodes can be utilized, and they cannot make an active choice in which node to communicate with.

5.3.4 Routing

On top of the assumptions mentioned earlier, nodes are also assumed to be placed with intent in the Tundra, much like a structured WSN. Therefore, a structure like figure 5.1 can be expected where relay nodes are centralized in the cluster. This does not necessarily need to be a required assumption due to messages always being able to reach any node through forwarding, meaning it could also be on the edge in scenarios where this might prove beneficial.

Because nodes does not ping each other, the system has to utilize pre-set paths between relays and normal nodes. This is done by having all relay nodes broadcast an election message. Each node receiving this message will rebroadcast and forward it until all other nodes has received it. While the message propagates, each node it passes through will append itself to the message so receivers learns the exact path to the relay. As there can be multiple relay nodes, this information is used to choose which election message to accept. To help the goal of minimizing message propagation, nodes will choose the path with the least amount of hops. It also currently discards any other possible path and only stores the final choice.

It is important to note that while it is named an election message, it does not bear much resemblance to a traditional election as it does not have to be accepted. A node's role is chosen before being placed and cannot be changed during run time. Instead, it simply makes itself available and informs other nodes of its presence.

5.3.5 Node Life-cycle

Due to how the routing is set up, a typical node life-cycle can be differentiated into two different phases; a setup phase and a main phase. The setup phase is where the relay nodes will broadcast their election messages. This phase is driven by the relay nodes and sets up the network state for the whole system by creating all the data paths between nodes. The setup phase is only entered a single time at the start of a node's life.

The main phase is where a node spends the majority of its time. This phase is driven by the normal nodes and is also the simplest phase. The only responsibility of the system in this phase is to ensure that nodes wake up, sends its data to a relay node and goes to sleep again. Currently, relay nodes will only wake up and listen for messages in this phase. However, as mentioned earlier, it can be equipped with sensors and store the data to its local storage as it does not need to report it to other relay nodes.

5.4 Design

5.4.1 Message Handling

There are two types of messages utilized by the system. The first type is an "election message" as mentioned earlier. This message is only sent by relay nodes. It includes a path of nodes it has passed through, the id of the current sender and the id of the original sender. The second type is a "data message". This message is only sent by normal nodes. It contains the path to the relay node as well as the sensor data to be delivered, along with the id of the original sender and the destination id. The reason this message contains the path is that it is dynamically updated as it travels. The message itself dictates where to travel and not the node it arrives at, as each node only knows its own path, which might differ from a neighbors' path.

Each node has a message handler which will check for the type of message and act accordingly. To ensure that messages aren't being sent around in loops indefinitely in the system, each message is assigned a unique id. Whenever a node receives a message, it will save this id. If it already has this saved, it simply drops the message. If it isn't saved, it will act according to the message type and either rebroadcast the election message, or send the data message to the next node in the path. Additionally, when the election message arrives, it will update its state accordingly if the saved path is longer than the new path.

5.4.2 Energy Model

E_elec	50 nJ/bit (Amount of joule per bit used by sender and receiver)
E_amp	100 pJ/bit/m^2 (Amount of power required by amplifier to transmit signal over a set distance)
K	Bits (Amount of bits used during transmission)
d^2	Distance (Squared distance in meters to transmit message)

Table 5.1: Variables used in equations for energy consumption

The energy model used is based on a combination between the LEACH[6] and the ESDS[22] papers. LEACH provides a model and numbers to calculate how much energy is expended to send a message over a distance, as well as how much is used when receiving a message. Equations 5.1 and 5.2 shows the formulas used for this. Table 5.1 contains explanations for each of the symbols. ESDS provides an assumption for how much energy is expended during a nodes up-time and estimates it to be 0.4 watts. The sum of these two are combined to provide a final count of energy expenditure.

$$E_{send} = E_{elec} * K + E_{amp} * K * d^2 \tag{5.1}$$

$$E_{recv} = E_{elec} * K \tag{5.2}$$

5.5 Implementation

5.5.1 Network and Node Grid

Server processes and a real network is not available due to the system being built on top of an event-based simulator. This means networking has to be simulated and is done by creating a class. Each node is registered in a list upon initialization. The class defines a broadcast and send API for nodes to utilize.

The system simulates nodes by placing them inside a grid. Each node has a defined set of x and y coordinates for their position, as well as a range vector to simulate the edge of their radio signal. This vector is the Euclidean distance between origo and the radio range in both the x and y direction. The grid size is assumed to be 500x500 meters to match the architecture. When utilizing the broadcast API, a node will iterate over the list in the network and calculate the Euclidean distance between itself and all other nodes. If this distance exceeds the range, the message is not sent. The same check applies to the send API, but it also matches based on node id.

5.6 Events

The system defines four different events; broadcast, send, sleep and wakeup. When a node tries to either broadcast or send a message, it creates an instance of the corresponding event and adds it to the event queue. The event itself then accesses the network class' API to perform the related action. The sleep and wakeup events simply toggle a sleep flag in the node, which is used to ensure that the send or broadcast event does not perform the action if a node is asleep.

As mentioned in the simulator chapter, a "less-than" magic method is required to ensure that no exceptions are thrown. This method can be used to create a priority for each event in the case of a duplicate timestamp. However, this is not utilized in the implementation. Adding such a priority can solve two problems. The first problem would be to prevent a node's actions from being executed in the wrong order. However, the burden of ensuring this ordering is on the user when defining its behavior.

The second problem is related to chaining events. As a consequence of multi-hop routing, when a send or broadcast event is executed, the receiving node's message handler will act accordingly and potentially create a new send or broadcast event. This means that a single event may end up creating multiple new events to be executed. To ensure that these are executed in the correct order, the given timestamp to the new event is slightly increased to simulate network delay. Due to how both of these problems are handled, there is no need for a priority among the event types.

5.6.1 Energy calculations

As mentioned earlier, the numbers used for energy calculations are based on ESDS[22] and LEACH[6]. This means that there is no need to simulate sensor data, as the size is part of the equations used. Therefore, the actual size of messages being sent between nodes is irrelevant. This solution is in part due to the actual messages being sent in the simulator being in the 100-300 bit range, which is vastly different from the 2000 bit size used.

The energy calculations are done in different parts of the code. The energy for receiving messages is incremented in the message handler. The energy spent sending a message is handled inside the actual event's execution. This is because the check for a node's sleeping status is also included there. This is done because a node cannot send a message while sleeping, and so the energy should not be expended. Finally, calculating the up-time is a two-part step which includes both the sleep and wakeup events. When a node wakes up its timestamp is saved, and when it goes back to sleep the amount of time awake is multiplied with the estimated watt usage to get the amount of joules. This is a viable solution as a node does not die in this implementation, otherwise this would have to be checked before and after any action is taken to ensure enough energy was present.

5.6.2 Messages

Messages are implemented using JSON and are passed between node instances through the use of events and the network class. As mentioned earlier, these messages do not contain large amounts of sensor data as it might have in a real system, but instead a simple test message for debugging.

The unique id used to prevent duplicates utilizes the uuid library from Python. This does not consider its efficiency, but it allows for all nodes in range to eventually get the message, compared to using a decreasing hop count as messages propagate through the network. For election messages, each node appends its id to the path list in the message. For data messages, each node removes the last id in the path before sending the message to the corresponding node. This allows the implementation to check for an empty path list to ensure that the message has arrived at the relay node.

5.6.3 Phase simulation

The current implementation of the system requires the two phases to be run two separate times through the event loop. As mentioned earlier, the event loop expects all events to be added to the queue before running. The reason for this is the implementation of the data message. It currently expects the path to be included in the message before being sent. However, without having ran the setup phase, this state does not exist. Therefore, each phase has to be simulated separately, but with its state being saved in between runs. This is handled in the main loop and does not require the complete thesis implementation to be run twice.

5.6.4 Node Runtime Behaviour

The current implementation provides a very simple API for defining a node's behavior. It is limited to four functions, where two of them are wakeup and sleep, one is for electing itself as a relay, and one is for sending sensor data. Listing 5.1 and 5.2 shows examples of the current implementation of each phase. Two functions are needed due to the implementation specific behavior around each phase. These functions are responsible for filling the queue with events before the event loop starts executing.

Specifically, the main function, which is responsible for the main phase, is heavily inspired from ESDS[22] and their examples. It performs a wakeup and send every hour. It is important to note that the delay between wakeup and send is given at least a second to ensure order. The send also happens in the first half of a node's uptime to ensure that this network delay does not end up dropping messages due to nodes falling asleep.

```
async def setup(self):
    await self.wakeup(0)
    if self.is_relay:
        await self.elect_self(1) # type: ignore

    await self.sleep(30) # All nodes sleep
```

Listing 5.1: Node Setup

```
async def main(self):
    for hour in range(1, HOURS+1):
        await self.wakeup(hour*3600)
        if not self.is_relay:
            await self.send_sensor_data(random.randint(
                hour*3600+1, hour*3600+int(UPTIME/2)-1))
        await self.sleep(hour*3600 + UPTIME)
```

Listing 5.2: Node Behvaiour

5.6.5 Configuration

To define the system setup itself and the positioning of all the nodes, a configuration written in YAML is utilized. This YAML file is passed as an input to the simulator and is read to create all the nodes. This makes it easy to set up various different clusters and allows for flexible testing. Each node contains an x and y position in the grid, as well as a flag to confirm if it's a relay node or not. An example of how the YAML file is structured can be seen in listing 5.3.

```
nodes:
  - id: 0
    x: 250
    y: 230
    relay: True

  - id: 1
    x: 270
    y: 260
    relay: False

  - id: 2
    x: 230
    y: 270
    relay: False
```

Listing 5.3: YAML configuration file

As mentioned in the simulator chapter, part of the configuration is handled through the config.py file. Some of these are specific to the system. This includes the network delay, how long a node spends awake, and how many hours the simulation should run. The latter one is very specific to the implementation of a node's behavior. Finally, the variables from the energy model can also be changed as needed, as well as the range of the node.

5.7 Version Numbers

Below is a list of the version numbers for all libraries and packages used. This is combined for the simulator and the data gathering system. Most of the items are fetched from the requirements file generated by pip.

- Python 3.8.10 (Including asyncio as part of the standard library)
- attrs 23.2.0
- cattrs 23.2.3
- contourpy 1.1.1
- cycler 0.12.1
- exceptiongroup 1.2.1
- fonttools 4.52.4
- importlib-resources 6.4.0
- kiwisolver 1.4.5
- matplotlib 3.7.5
- numpy 1.24.4
- packaging 24.0
- pillow 10.3.0
- pyparsing 3.1.2
- python-dateutil 2.9.0.post0

- PyYAML 6.0.1
- six 1.16.0
- typing-extensions 4.12.0
- zipp 3.19.0

6 Evaluation

This chapter starts with presenting the methodology and goal of both experiments that were run. It then presents the results for each of them.

6.1 Environment

As has been explained through this thesis, the system has been implemented on top of an event based simulator. This simulator has been run on an HP Envy x360 laptop. It features a 2.3 GHz processor with 4 cores and 16 GB of memory.

6.2 Experiment 1: Energy Expended in real systems

6.2.1 Methodology

This experiment is performed by setting the behavior of nodes to wake up once every hour, send its sensor data, and then go to sleep. It is executed on two different setups, which can be seen in figures 6.1 and 6.2. The first figure showcases a dense setup with many nodes close together. The second figure showcases a sparse setup with fewer nodes. These setups are assumed to use a 500x500 meter grid, and each node has a 100-meter range for their radios. Three variations are run; 24, 720 and 8760 hours (or 1 day, 30 days and 356 days) for both.

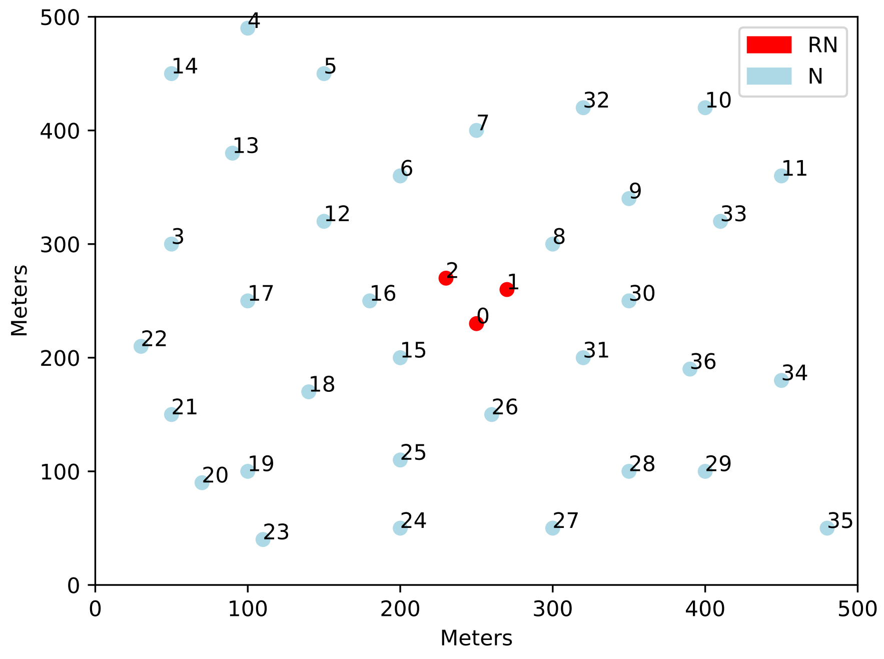

Figure 6.1: Position of all nodes in a dense setup. RN = Relay Node, N = Node

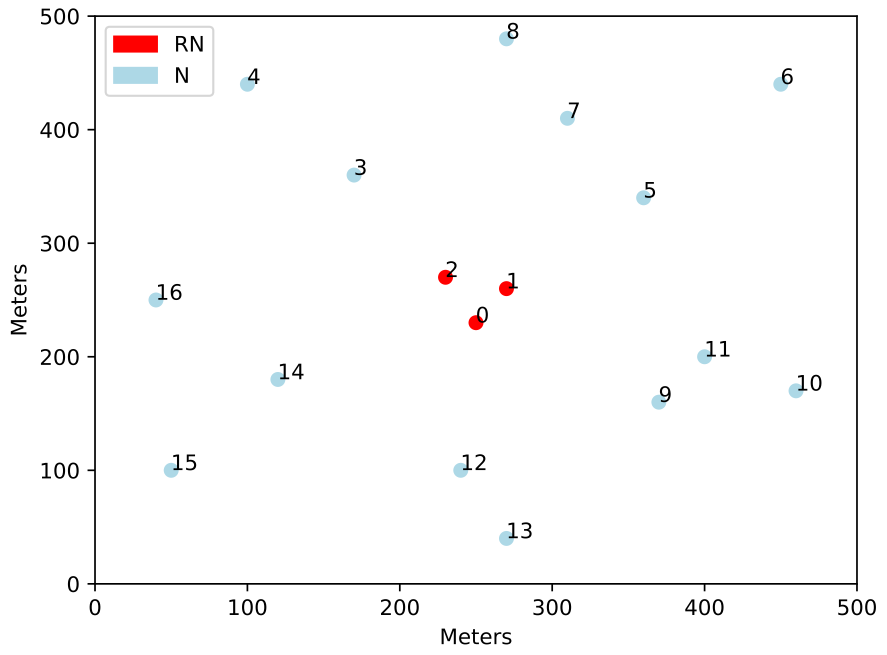

Figure 6.2: Position of all nodes in a sparse setup. RN = Relay Node, N = Node

6.2.2 Goal

The goal of this experiment is to showcase the energy expended by each node in the system. The following metrics will be presented:

- Energy in joule.
  - Total Energy.
  - Energy spent idle (meaning it is awake, but not doing anything).
  - Energy spent sending messages (the specific action of sending).
  - Energy spent receiving message (the specific action of receiving).
  - Energy expended while asleep is assumed to be low enough to be ignored.

6.3 Experiment 2: Simulator Scalability

6.3.1 Methodology

This experiment is also performed by setting the behavior of nodes to wake up once every hour and send its sensor data before going to sleep again. It is executed on a set of systems organized in a line, which can be seen in figure 6.3. The actual experiment tests systems with 1-5, 10 and 20 nodes. The lower count systems simply shorten the line. All experiments are only run for 1 hour. This experiment will also present the relevant data from the setups run in the previous experiment.

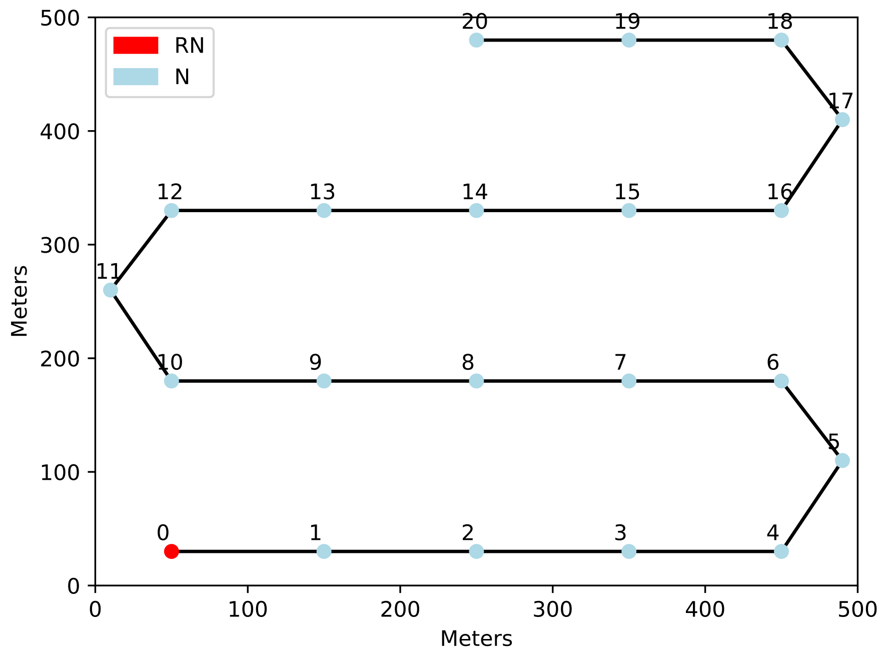

Figure 6.3: 1 Relay node and 20 Normal nodes connected together. RN = Relay Node, N = Node

6.3.2 Goal

The goal of this experiment is to showcase the scalability of the simulator. The following metrics are used:

- Time spent running simulator.

- Event Count.

6.4 Results

6.4.1 Experiment 1: Energy expended in real systems

	Mean	Standard Deviation
Energy	588.169 J	0.115 J
E_Recv	0.00605 J	0.00937 J
E_Send	0.164 J	0.115 J
E_Idle	588.0 J	0.0 J

Table 6.1: Dense setup 24 Hours, Mean and Deviation for all Nodes

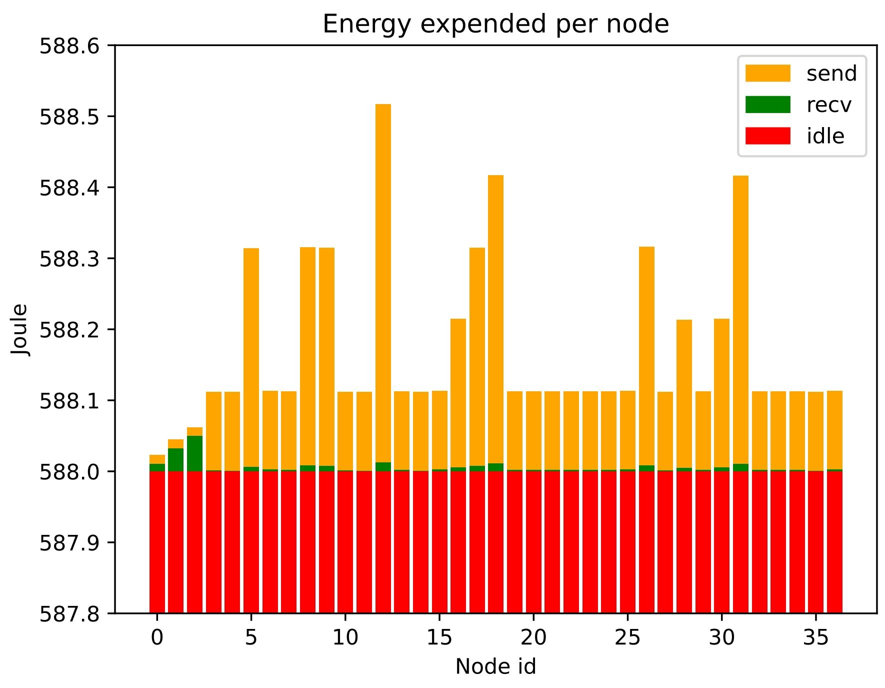

Figure 6.4: Dense setup 24 Hours. Y does not start at 0.

Figure 6.4 and table 6.1 shows the energy expended for a dense system running for 24 hours, note that the y-axis does not start at 0. On average, nodes spent 588 J being idle with a standard deviation of 0.12 J. 0.006 J on sending messages with a standard deviation of 0.009 J. And 0.16 J on receiving messages with a standard deviation of 0.12 J.

	Mean	Standard Deviation
Energy	17296.673 J	3.448 J
E_Recv	0.113 J	0.264 J
E_Send	4.560 J	3.451 J
E_Idle	17292.0 J	0.0 J

Table 6.2: Dense Setup 720 Hours, Mean and Deviation for all Nodes

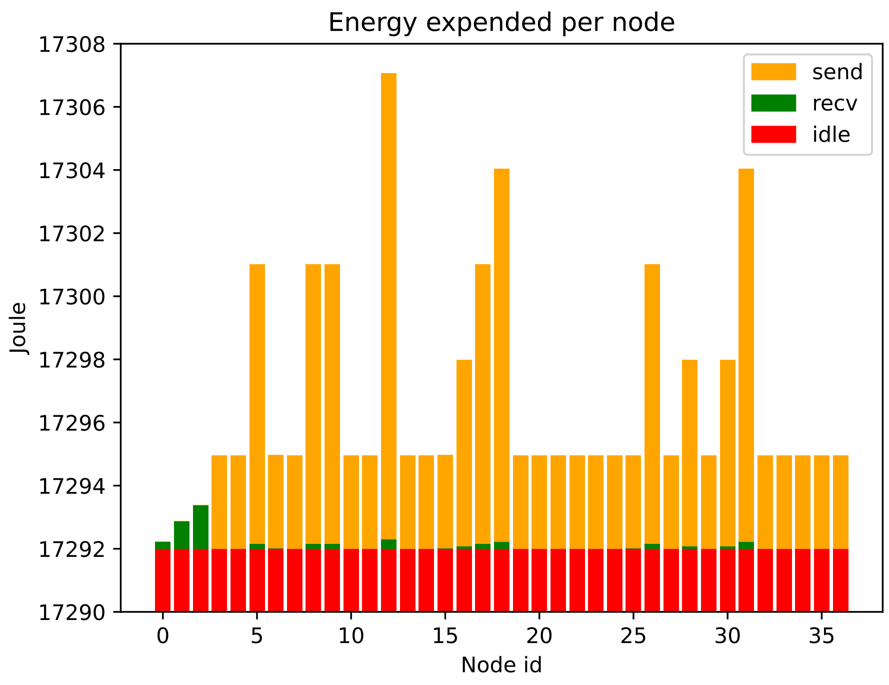

Figure 6.5: Dense setup 720 Hours (1 Month). Y does not start at 0.

Figure 6.5 and table 6.2 shows the energy expended for a dense system running for 720 hours, note that the y-axis does not start at 0. On average, nodes spent 17297 J being idle with a standard deviation of 3.45 J. 0.11 J on sending messages with a standard deviation of 0.26 J. And 4.56 J receiving messages with a standard deviation of 3.45 J.

	Mean	Standard Deviation
Energy	210308.694 J	41.943 J
E_Recv	1.352 J	3.209 J
E_Send	55.342 J	41.983 J
E_Idle	210252.0 J	0.0 J

Table 6.3: Dense setup 8760 Hours, Mean and Deviation for all Nodes

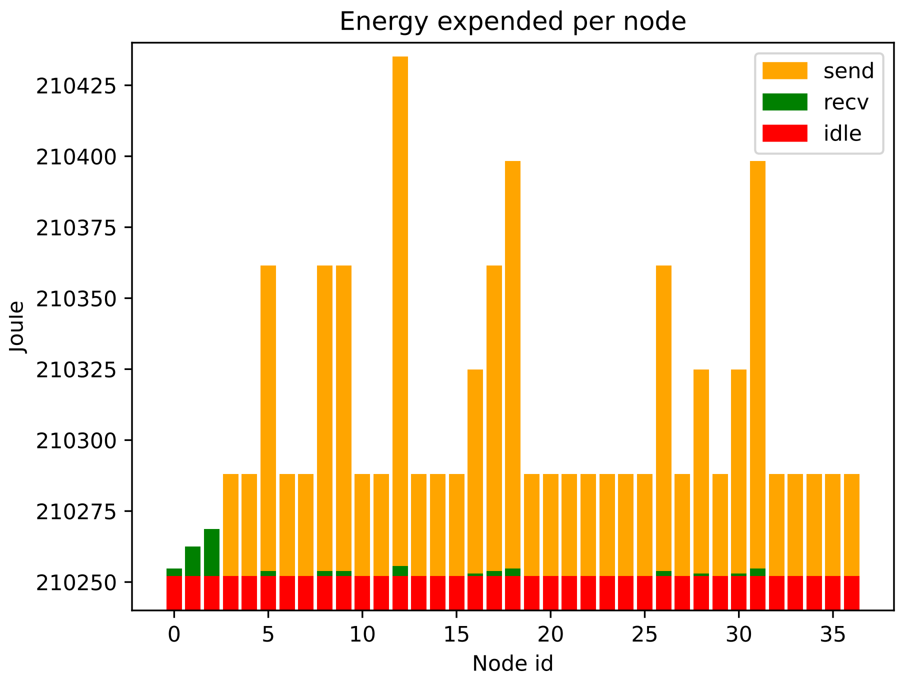

Figure 6.6: Dense setup 8760 Hours (1 Year). Y does not start at 0.

Figure 6.6 and table 6.2 shows the energy expended for a dense system running for 8760 hours, note that the y-axis does not start at 0. On average, nodes spent 210308 J being idle with a standard deviation of 41.94 J. 1.35 J on sending messages with a standard deviation of 3.21 J. And 55.34 J receiving messages with a standard deviation of 41.98 J.

	Mean	Standard Deviation
Energy	588.155 J	0.110 J
E_Recv	0.004 J	0.005 J
E_Send	0.151 J	0.110 J
E_Idle	588.0 J	0.0 J

Table 6.4: Sparse setup 24 Hours, Mean and Deviation for all Nodes

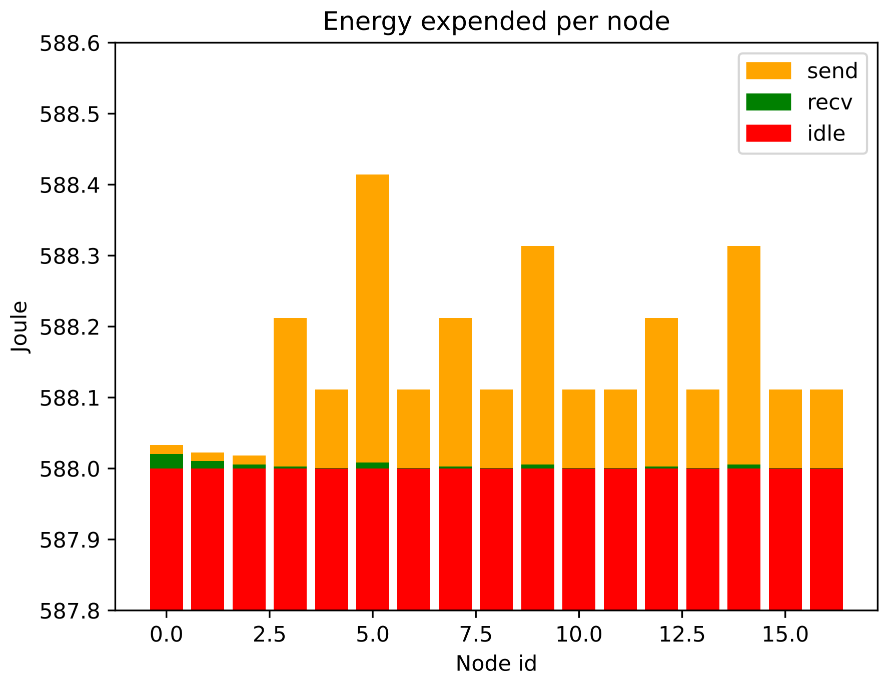

Figure 6.7: Sparse setup 24 Hours. Y does not start at 0.

Figure 6.7 and table 6.4 shows the energy expended for a sparse system running for 24 hours, note that the y-axis does not start at 0. On average, nodes spent 588 J being idle with a standard deviation of 0.11 J. 0.004 J on sending messages with a standard deviation of 0.005 J. And 0.15 J on receiving messages with a standard deviation of 0.11 J.

	Mean	Standard Deviation
Energy	17296.282 J	3.297 J
E_Recv	0.102 J	0.151 J
E_Send	4.179 J	3.310 J
E_Idle	17292.0 J	0.0 J

Table 6.5: Sparse setup 720 Hours, Mean and Deviation for all Nodes

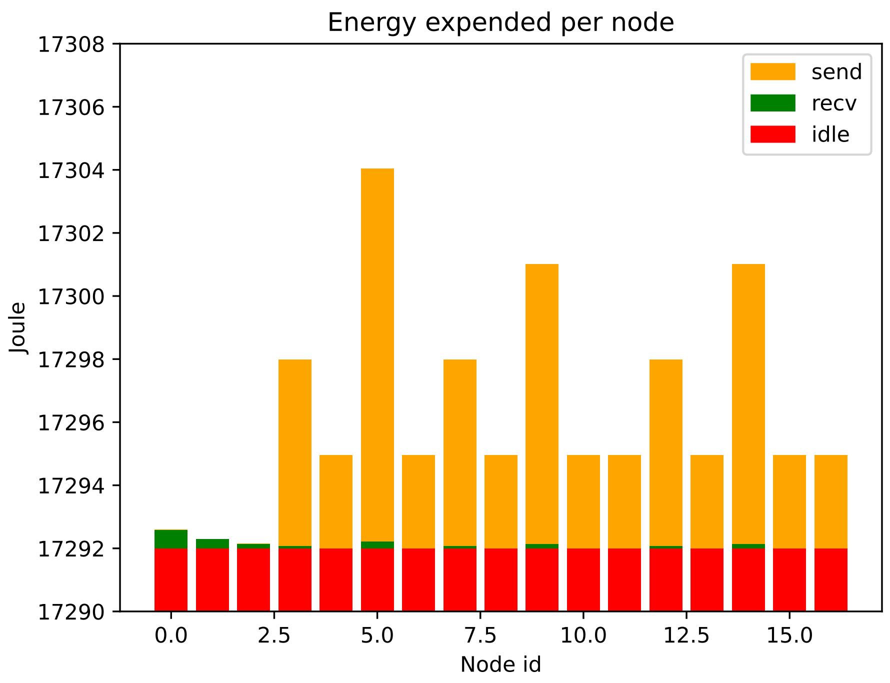

Figure 6.8: Sparse setup 720 Hours (1 Month). Y does not start at 0.

Figure 6.8 and table 6.5 shows the energy expended for a sparse system running for 720 hours, note that the y-axis does not start at 0. On average, nodes spent 17296 J on being idle with a standard deviation of 3.30 J. 0.10 J on sending messages with a standard deviation of 0.15 J. And 4.18 J on receiving messages with a standard deviation of 3.31 J.

	Mean	Standard Deviation
Energy	210303.954 J	40.125 J
E_Recv	1.237 J	1.834 J
E_Send	50.717 J	40.273 J
E_Idle	210252.0 J	0.0 J

Table 6.6: Sparse setup 8760 Hours, Mean and Deviation for all Nodes

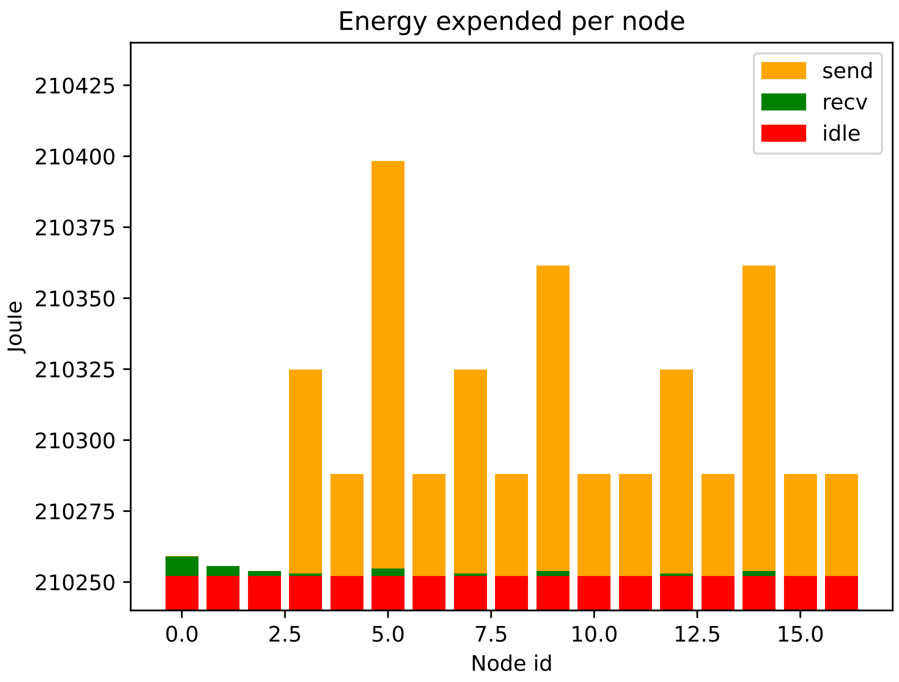

Figure 6.9: Sparse setup 8760 Hours (1 Year). Y does not start at 0.

Figure 6.9 and table 6.6 shows the energy expended for a sparse system running for 8760 hours, note that the y-axis does not start at 0. on average, nodes spent 210304 J being idle with a standard deviation of 40.13 J. 1.24 J on sending messages with a standard deviation of 1.83 J. And 50.71 J on receiving messages with a standard deviation of 40 J.

6.4.2 Experiment 2: Simulator Scaling

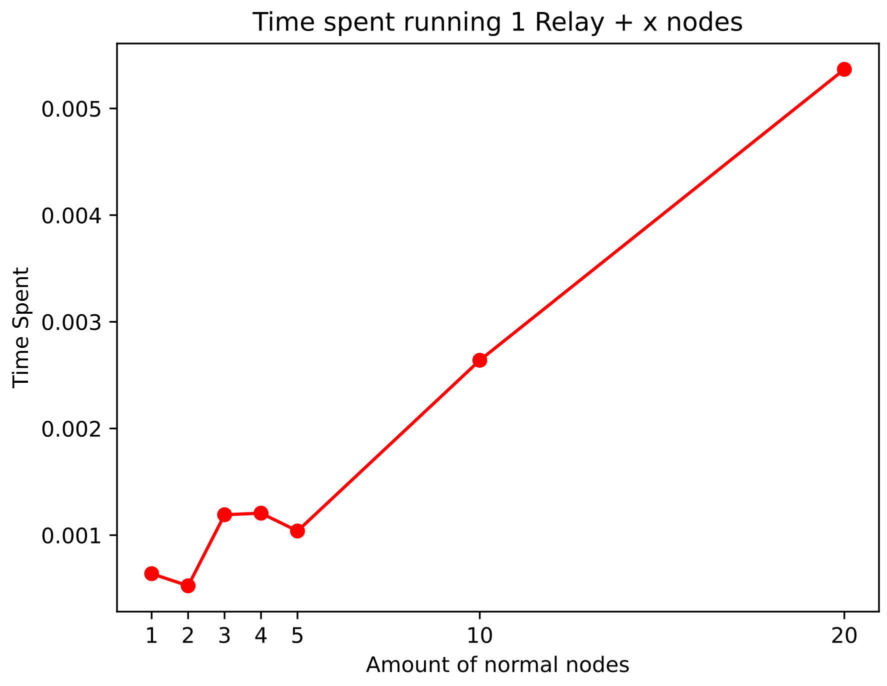

Figure 6.10: Timer Scaling

Figure 6.10 displays the timing results of the experiment. For the lower end there are small jumps before a more distinct line is formed between 5, 10 and 20 nodes.

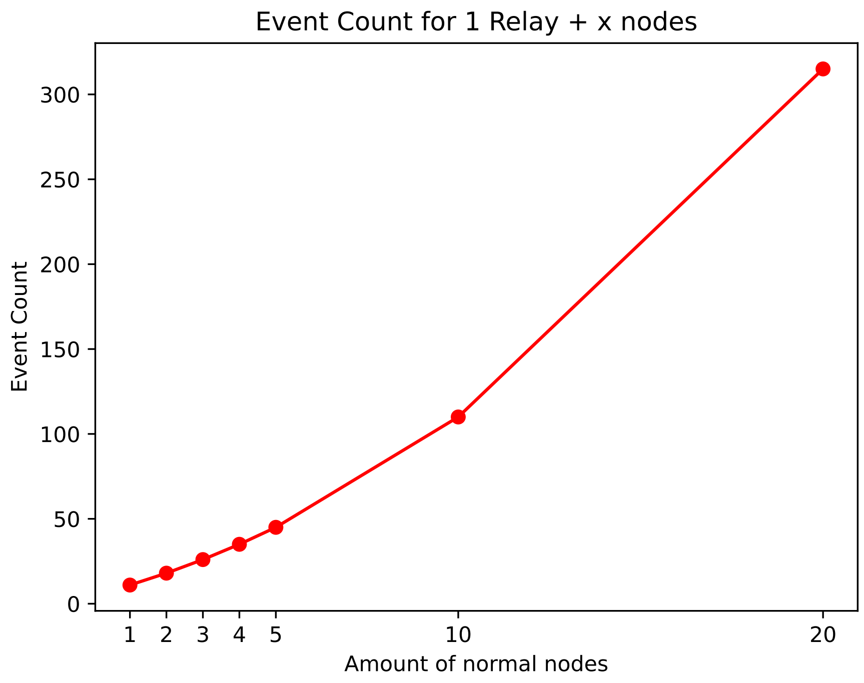

Figure 6.11: Event Scaling

Figure 6.11 displays the event count results of the experiment. It shows a non-linear increase in the amount of events as more nodes are added.

	Timer	Event Count
Dense 24h	0.491 s	3329
Dense 720h	17.920 s	94505
Dense 8760h noprint	1239.819 s	1147745
Dense 8760h print	1735.734 s	x
Sparse 24h	0.181 s	1477
Sparse 720h	6.047 s	41845
Sparse 8760h noprint	150.145 s	508165
Sparse 8760h print	259.504 s	x

Table 6.7: Timer and Event Count from Experiment 1

Table 6.7 displays both the event count and the timers from the first experiment as an addition to the results of the second experiment. Simulating a dense system for 24, 720 and 8760 (without/with print) took 0.5, 17.9, 1239 and 1935 seconds. Simulating a sparse system for the same hours took 0.1, 6, 150 and 259 seconds. For the 1 year long simulations this equates to about 20 and 29 minutes for the dense setup, and 2.5 and 4 minutes for the sparse simulation.

For the event counts this equated to 3329, 94505 and 114775 events in the dense system, and 1477, 41855 and 508165 events. Printing has no effects on the event count.

7 Discussion

This chapter first discusses the results of the experiment measuring the energy consumption of nodes. It covers the effect of messages having a low cost and how mesh could be utilized instead. It also looks at how the dense setup fits mesh networking, and the sparse setup fits the current system. Then, the results from the scaling experiment is discussed, before various topics such as being part of an intermediary path, dead paths, implementation problems, system coupling and a few more are covered.

7.1 Message Energy Cost

One of the most notable results of the first experiment in the evaluation section is how the energy consumption is split between idle up-time, sending, and receiving messages. With the current energy model, almost all of the energy is spent by a node being awake. This can be decreased by lowering the up-time of a node. The problem with doing that has to do with the abstracted away synchronization. Lower up-time means less time for nodes to do their work, and increasing the chance that a node goes to sleep before others can finish. This is a very important trade-off, and the up-time has been set to at least 60 seconds to account for this. Abstracting away synchronization makes it simpler to simulate, but it opens up the possibility of minimizing up-time to the extreme, as it does not need to account for clock skew.

Another consequence of the majority of the energy consumption being from idle up-time is that sending messages is very cheap. Therefore, the intended goal of minimizing the amount of messages sent is actually less important than expected, as the energy consumed is far below what was assumed. This opens up many possibilities for improving the system, one of the main ones being ping messages. This allows each node to obtain information about the overall network state, as well as the state of neighboring nodes. The information can then be used by more complex routing protocols to further optimize the network traffic and fix the problem of dead paths, which will be discussed later in this chapter. However, it is still important to note that there are limited network capabilities, and it may still stand out as a reason to utilize this approach.

7.2 Mesh Networking

When minimizing messages is no longer crucial, it opens up the possibility of mesh networks, as nodes can now ping and update each other. Limited network capabilities is also one of the use cases for mesh, to widen a network's range. The advantage of using a mesh network is that assumptions where each node must be able to see the whole network can now be fulfilled. This is done by each node thinking it has a direct connection to all other nodes, but instead there is an underlying protocol forwarding messages through other nodes to reach its destination. This allows for many of the well researched protocols such as LEACH[6] or PEGASIS[19] to be implemented and run in the Arctic Tundra. It could be very interesting to see what effect this has on the energy levels in the system.

Mesh networks can be a good fit if the wireless sensor network is similar to the dense setup used in the experiments. Figure 7.1 shows how the dense configuration looks when a line is drawn between all nodes within range of each other. Each node has multiple routing options, and it can therefore prevent scenarios where a single node is responsible for forwarding messages for many others. This means that there is not a single point of failure in the routing. For comparison, figure 7.2 shows how the current algorithm sets the paths. Uneven battery drain can also be prevented by carefully selecting the mesh routing algorithm. An option could be one where each node builds the whole network state in memory, or one where multiple paths between nodes are saved.

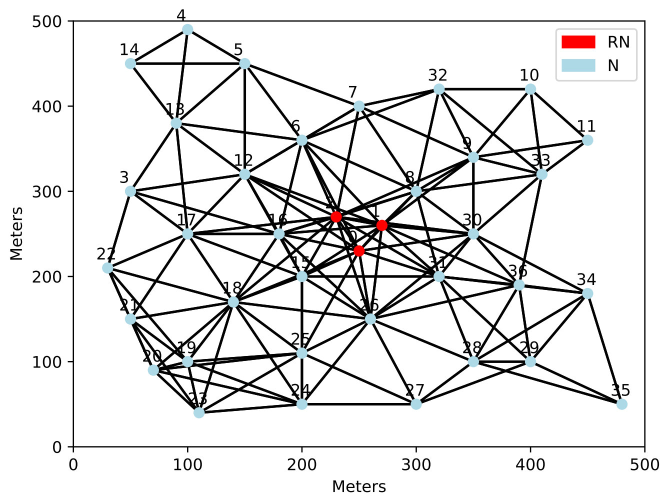

Figure 7.1: Dense configuration with all possible paths. RN = Relay Node, N = Node.

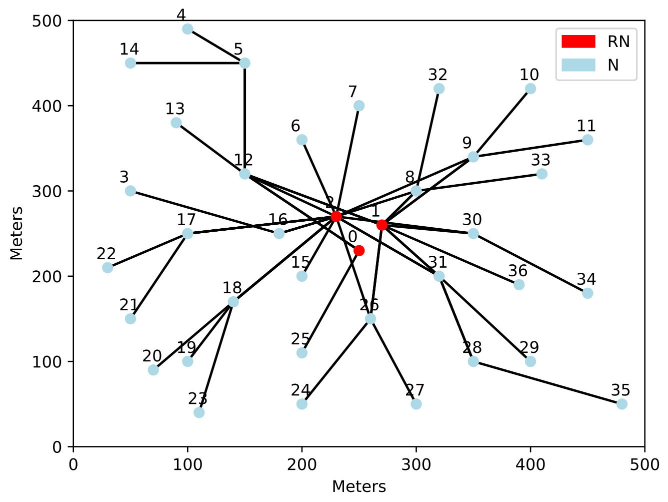

Figure 7.2: Dense configuration with chosen paths. RN = Relay Node, N = Node.

7.3 Network setup fitting the current system

Drawing all possible paths for the dense system shows that mesh could be a very good fit for sensor networks with many nodes. By doing the same with the sparse setup, each node has a lot less options when communicating. When comparing figure 7.3 and 7.4, there are only a few paths removed. Fewer nodes spread out over larger distances obviously means less options for routing, or even only a single path being available, as is the case for the sparse setup. Therefore, we can conclude that the system this thesis has proposed is a good fit for a sparse system, as many of the connected paths only have a single point of failure anyways. However, if it is utilized in a dense setup, the other paths are not utilized and it can be seen as a worse fit.

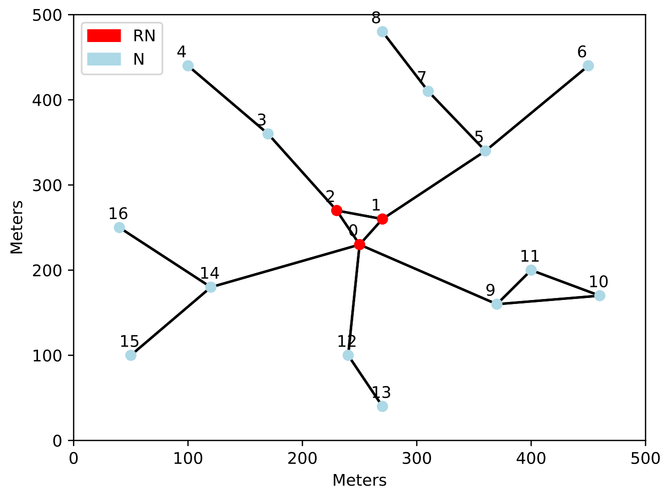

Figure 7.3: Sparse configuration with all possible paths. RN = Relay Node, N = Node.

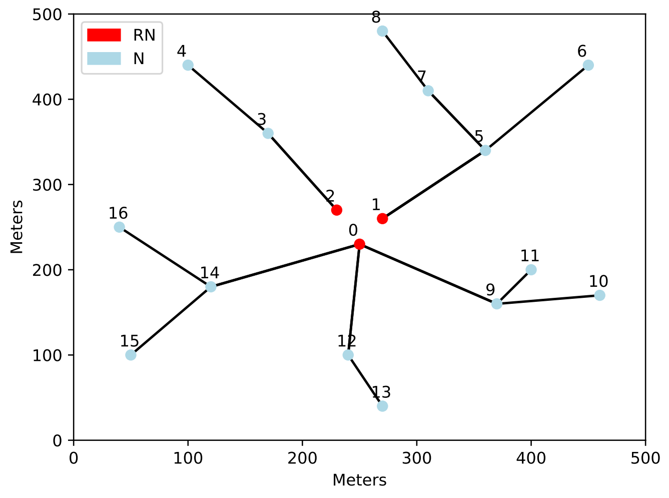

Figure 7.4: Sparse configuration with chosen paths. RN = Relay Node, N = Node.

7.4 Simulator scaling

The second experiment covers both run time and event scaling. Figure 6.10 shows that there is no real pattern in scaling for time spent simulating in smaller systems. The possible problem here is that the run-time is so small that underlying factors such as performance drift in hardware affects the results. This could be due to other programs running, or bad luck in OS scheduling where it prioritizes other processes. However, there is an increase in time spent simulating the system, but it does not have a recognizable pattern. As such, there is no probable conclusion to be drawn from this experiment other than time increasing as the system size increases.

Before continuing, it is important to note that all simulations were run while printing each event being fetched from the queue. This behavior is adopted from ESDS[22] as they also do the same. It is relatively common knowledge that IO operations (including printing) are slow in most programming languages, and especially Python. Therefore, the one-year-long simulations were also run without any prints to showcase the time it could save. For example, a dense setup goes from 29 minutes to about 20 and a half minute, and a sparse setup goes from about 4 minutes to about 2 and a half minutes.

A better example to explore how the simulators running time scales is by looking at the data from the dense and sparse configurations used in experiment 1. First of all, an increase from 24 hours to 720 hours is 2900%, and from 720 hours to 8760 hours is about 1117%. In comparison, the dense configuration had an increase of 3553% from 24-720 hours and 9696% from 720-9760 hours with prints, and 6826% without prints. For the sparse configuration, there was an increase of 3255% from 24-720 hours and 4196% from 720-9760 hours with prints, and 2385% without prints.

These percentage increases shows that there is no one-to-one correlation between increasing the simulated time and the time spent simulating. It only tells us that there is a non-linear increase greater than 1 as all steps has a bigger increase in time spent than the increase of hours simulated.

For events, figure 6.11 shows that there is an increase similar to exponential (note that it is not an exponential graph, but this type of shape is often mentioned as exponential in general). Looking at percentages from experiment 1 again, the dense configuration has an increase of 2744% from 24-720 hours, and 1114% between 720-8760 hours. The sparse configuration has an increase of 2744% for 24-720 hours, and an increase of 1115% from 720-8760 hours. This shows us that the correlation between hours simulated and event count has a very similar increase. This is expected behavior due to how the experiment is set up, where each node sends a single message per hour. Therefore, adding more hours adds a set amount of new events to process. This also means that there is no difference in percentage increase between a dense configuration increasing from 24 to 720 hours, and a sparse configuration doing the same.

To summarize, there is no recognizable pattern for timer scaling, but event scaling is similar to exponential. However, when looking at the values themselves, we see the benefit of using an event-based system. A sparser system can simulate a year over 4 minutes, and a denser system can simulate it over 29 minutes. It is to note that a dense system has 37 nodes and a sparse has 17, so there is a significant time increase when increasing the node count. However, this is still a lot of time saved when trying to evaluate certain numbers over long periods, or ensuring that the protocol seems to operate as expected.

It is important to note that bugs in hardware and underlying software such as operating systems may happen, and small implementation errors may occur. This kind of information can only be obtained by actually implementing and deploying the system in the wild, which is one of the downsides of simulation.

7.5 Effect of being an Intermediary Path Node

In the dense configuration, some nodes ends up being a part of the path for multiple edge nodes. To see what effect this has, the nodes with id 18 and 15 can be compared. Node 18 is an intermediary for three other nodes, whereas node 15 is connected directly to the relay. By looking at the resulting file for a run, we can get the amount of messages each node has sent and received, although this is not presented part of the results in the previous chapter. Node 18 has sent 99 messages and received 111, and node 15 has sent 27 and received 30 over a 24-hour period. These numbers make sense as it sends out 24 data messages and rebroadcasts 3 election messages while also receiving it from 10 other nodes in range.

Looking at the energy levels, node 15 spent 588.113 J total and node 18 spent 588.416 total. Both spent 588 being idle, as such there is only a difference of about 0.3 J. Node 18 spent 0.011 J on receiving 111 messages and 0.4 J on sending 99 messages. Node 15 spent 0.0029 J on receiving 30 messages and 0.11 J on sending 27 messages. This is a difference of about 0.17% and is essentially negligible with the current energy model, which further supports the point that the goal of minimizing messages is not an important focus. The total energy difference between the nodes is about 0.05%. This difference stays the same for both 720 and 8760 hours of simulated time as well.

7.6 Dead Paths

One of the major problems with the current implementation is that there is no fail-safe in how it handles the routing. The setup phase is only run once at the very start of a node's lifetime, and there are no recovery options. For example, if an intermediate node dies it does not try to rebuild the path, and any of the alternate paths it has seen are already discarded. This essentially means that each path has a single point of failure, multiple nodes will be affected, and a significant loss of sensor data may occur. It also means that it will suffer from the same problems as a sparse system, where it essentially only has a single path despite being a part of a dense system with many nodes close by.

Additionally, the system does not use acknowledgements, as it assumes radio signals will always reach its target if in range. Therefore, a node has no way of knowing if any of the nodes in its path has stopped working. Furthermore, each node has chosen an individual path, so these acknowledgements would need to be propagated to the sending node. This is a problem specific to this approach, as it will continue to try and send messages even if nodes further in its path are dead. This problem could be circumvented by trying to rebuild paths through neighboring nodes. However, this is a problem because it does not necessarily know if that node can reach a relay node.

7.7 The problem with creating all events before running the queue

The current implementation of the simulator fills up the event queue before execution. However, there are a few obvious problems with this approach. The first problem is having to split up the setup phase and the operation phase into two separate executions of the event queue. As mentioned earlier, the current implementation took inspiration from ESDS[22] in how it allows a user to define a node's behavior. ESDS evaluates this function during run time and inserts events accordingly. It prevents a premature exit by defining a set of restrictions to ensure that each node completes before the event queue is empty. The current implementation is a result of trying to circumvent the complexity needed for such an approach, and this problem is a byproduct of that.

The second problem is related to what you can test with the system. The current implementation makes it harder to run a simulation until a node dies. For example, LEACH[6] provides results showing the amount of rounds completed until the first node is out of energy. The current implementation has no way of implementing this, as the exit condition cannot be stated in the node's behavior. Instead, this has to be done in the simulator or system itself by evaluating a node's energy every time an event is executed.

7.8 System and Simulator Coupling

Related to this is the problem that the simulator is too specialized, meaning it is highly coupled with the implemented system on top. The implementation is completely tailored around the system. This is not necessarily a bad thing in itself, but testing other solutions becomes hard, if not impossible. The current routing algorithm in the system is based on actions a node does when receiving a message. This makes it hard to test something like a mesh network, as it would require a significant rewrite of both system and simulator logic to allow for this. This is under the assumption that the network class is deemed as part of the simulator and not the system. While I allude to this being a problem, it might also be an optimal approach. It might just need a very specifically defined API for each component and what is expected to happen whenever any parts of this API is called.

7.9 Why simulating the Arctic is hard

The problems discussed in the sections above are part of the reason why simulating a system in the Arctic Tundra is so hard. First is what abstraction level to approach the problem from. Certain assumptions have to be made, and a choice must be taken for how fine-grained elements such as networking is going to be. Then the system itself has to be designed. Some of the approaches are harder to implement on top of certain simulators than others, if not impossible. This is due to each simulator having to make these assumptions and target certain abstraction levels. As an example, NS3 and OMNeT++ are both packet level simulators with a focus on networking. Simpy on the other hand, is a completely generalized simulator which can be utilized for many other things requiring discrete event simulation. You have to utilize its resource abstractions to create functionality such as broadcasting between nodes.

7.10 Asyncio

A major implementation component that can be criticized is the use of Asyncio. The only part of the system actually utilizing it is the event loop and the priority queue. However, the priority queue is implemented on top of the standard library heapq. There is no need for asynchronous or threaded programming in this implementation as each node does not act as its own mailbox, but instead is just state in memory which is changed as events are executed. Therefore each node does not need its own thread as the event itself executes the action, and the timestamps in the queue drives the simulation forward, so there is no need to wait for other nodes to finish operating.

7.11 Lowest length vs fewest hops

This system always tries to choose the shortest path in terms of fewest hops between nodes. However, this does not necessarily mean it travels the shortest distance. For example, a node could choose a path where the next node is 100m away as it only sees one node in the path, instead of a node which is 50 meters away with a single hop in between. To see which of these approaches works best, we can compare how much energy is expended when a message is sent 100 meters at once versus 100 meters with one or two hops.

Three scenarios are defined: 1x send for 100m and 1x receive, 2x send for 50m and 2x receive, and 3x send for 33m and 3x receive. By plotting these values into the energy model described in chapter 5 (Data Gathering System), we get the following values; 0.0022 J for one send and receive traveling 100m, 0.0014 for two sends and receives traveling 50m each, and 0.00116 J for three sends and receives traveling 33m each.

This shows that it is actually more beneficial energy with the current model to focus on traveling a shorter distance instead of focusing on fewer messages being sent. Figure 7.5 shows that there is an exponential increase in energy costs, meaning that an increase in range will always have an even bigger increase in energy costs. Additionally, receiving messages is quite cheap, which can be seen in the examples where even low ranges such as 33 meters can save energy.

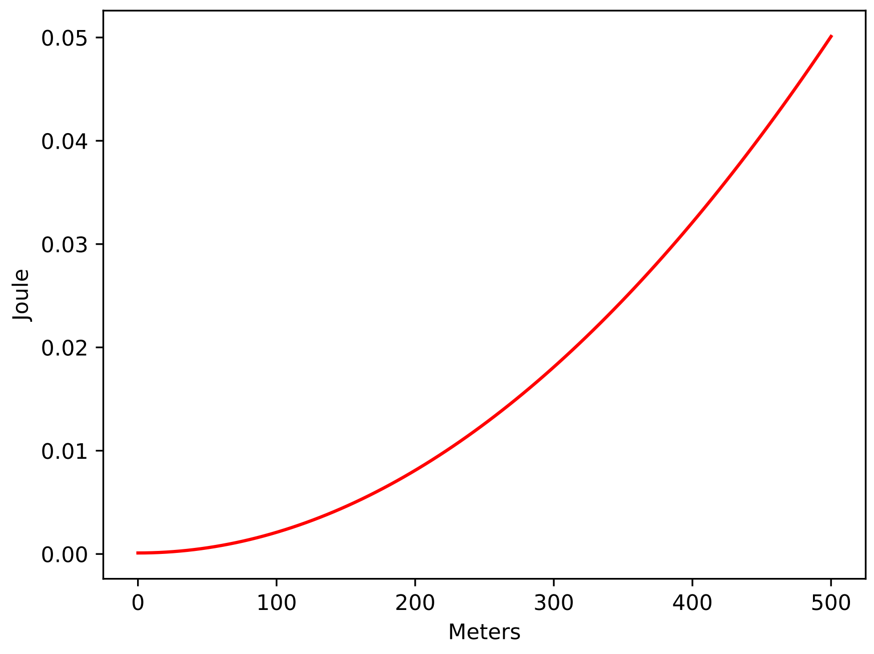

Figure 7.5: Energy scaling for sending messages over a distance.

7.12 Hardware evolution

When designing this system, a big assumption was made where radios were assumed to have limited range, and a lot of the inspiration was taken from LEACH[6]. However, LEACH is a paper from the year 2000, and massive leaps has been made within radio hardware. An example of this is LoRa radios. While the actual range is very reliant on the environment, it is assumed to be able to reach a few kilometers in urban areas, and even longer in rural areas.

This could be very interesting to try and utilize in the Arctic Tundra. This means that utilizing algorithms such as LEACH or any of the newer variations could be a real possibility within cluster in this system. It eliminates the need for multi-hop routing and allows for single hop and direct protocols to be utilized. One of the big pros of this is that these algorithms moves complexity away from routing.

Another set of hardware which could be interesting to test out is using micro-controllers operating on the ZigBee standard[23]. It is a networking standard aimed at low power devices which utilizes mesh networking to ensure reliability. Perhaps comparing these could be interesting.

7.13 Node Placement

It is important to note that this thesis and implementation only operates in a single 2D plane along an imaginary ground. However, real nodes also operate in a second plane, including height. This introduces the challenges of some nodes potentially being underground, in ditches, or on top of mountains. Additionally, there might be "radio shadows", which are small zones where the signal might not reach. An example of this is often when a sensor is behind a dense rock, or covered by bushes and trees. These are problems this thesis has ignored, and despite new hardware being better, these problems might still be very relevant. Therefore, research on multi-hop routing in this context is still very important. Perhaps a combination of multi-hop and single-hop routing is one of the options to solve this, where it tries direct communication first if possible, and then falls back on forwarding through other nodes if not.

8 Future Work

This chapter discusses some of the future work can be done. It first presents some research paths, before mentioning some implementation improvements for the system.

8.1 Mesh Network

The discussion section has already covered this to an extent, but one of the possible ways forward is to start exploring systems utilizing mesh networks. They are one of the better ways to increase network size and allow for communication between nodes not in range of each other. One of the important areas to explore is the energy consumption of nodes and looking for bottlenecks where some nodes might be a massive hinder for the system.

8.2 Energy Models

This thesis chooses an energy model which is a simple combination between two proposed models from other research papers. A direction to take both this thesis or perhaps other research in general is to look at various energy models and compare it to actual documented energy consumption. Additionally, the low temperatures in the Arctic Tundra might have a significant effect on battery lifetime and this could be important to include in the model itself. Therefore, spending more time looking into various models is important as it will lead to more accurate results.

8.3 Hardware

One of the assumptions this thesis might have gotten wrong from the start is the assumptions about hardware. Radios or alternate communication options have seen large improvements, which is not apparent in research papers from 20 years ago. Therefore, even in the Arctic, the assumptions of nodes being able to reach all other nodes could be viable. Testing out algorithms such as LEACH or PEGASIS is an interesting approach to take for the Arctic itself. However, this may only be true inside clusters. Some form of stronger communication device might be required to communicate between clusters and the base station, similar to the approach of LEDHE-WSN[18].

8.4 Implementation Improvements

There are a few improvements which can be done to this thesis if seen from the lens of a software engineer. The first one is to replace the use of unique ids with a form of sequence numbers instead. The reasoning for this is that a unique id might end up being such a large integer that its size ends up being a significant part of the message header, where it might even be bigger than the payload. Utilizing sequence numbers will require tackling a new set of problems. One of these is ensuring messages are dropped, but this could be handles by utilizing hop-count and ensuring a message only has a limited travel time. Another option is to combine the node id with a message count. This will work in the short term for systems sending fewer messages, but it might end up problematic in networks with massive amounts of nodes or very long life times.

Another part that can be improved is to better define the APIs. For example, what kind of options should the network class expose to a node, and how does a node register in the network. This means closely defining the behavior which a node expects. One of the limitations in this implementation is that it is fairly coupled and doesn't allow for easy testing of alternate implementations. The current network class exposes a send and broadcast API, but should also include a register function.

This can also be included for the node itself. This is something that ESDS[22] does well in its paper. A node has a very specific API it can utilize, which is presented to a user. The system in this thesis does not expose APIs with this in mind, and as such it might be confusing for a user to change the behavior of a node. This is in addition to the two phases having to be executed separately, so the behavior has to be defined twice.

Therefore, another improvement which can be done is to solve the problem of the event queue. This might require an approach more similar to ESDS where nodes are evaluated in real time and each thread fills the event queue and waits. Threading is generally considered a heavy operation, so it could be interesting to try and do this with asyncio instead by utilizing async/await and yielding control of the main thread.

/9 Conclusion

This thesis has presented the design for a data gathering system which aims to tackle some of the networking challenges that wireless sensor network algorithms face in the Arctic. Additionally, challenges with simulating such systems has been explored, which resulted in building an event-based simulator to study the proposed design. The main challenge the system aims to overcome is nodes having limited range and being unable to form direct connections to all other nodes. Some assumptions have to be made; nodes are always synchronized, and can reach any other node through forwarding. The complete architecture contains a base station and multiple clusters, but due to time constraints, this thesis only focuses on the internals of a single cluster.

The design aims to minimize messages due to energy consumption and constrained networks, and therefore does not ping other nodes. This causes each node to have a very limited view of the network. Two types of nodes are defined, a normal node which gathers data, and a relay node which aims to communicate with a base station. Two phases are defined; a setup phase where relays broadcast an election message to inform other nodes of its presence, and a main phase where normal nodes send their data to the relays. Paths are formed in the setup phase and are a part of the data messages to help each node forward correctly, eventually causing sensor data to reach relay nodes.

Two experiments were run, one to evaluate the energy consumption of the system, and one to evaluate the scalability of the simulator. The results found that idle up-time consumes the most energy, and messages are only a small part of the total consumption. This means that more complex routing protocols can be utilized. Specifically for dense systems, an approach using mesh networks seem appropriate. On the other hand, the proposed system is deemed to best fit sparser systems where nodes do not have many, if any, alternate paths. Additionally, the scalability results showcase that there is a non-linear increase in time spent compared to hours simulated, but it doesn't compare to any specific model. Event scaling on the other hand is similar to an exponential increase, which makes sense due to how the experiment is run.

To summarize, this thesis has contributed with the design of a data gathering system, as well as implementing it on top of an event-based simulator to study the proposed system. A discussion around the results, the suitability of the system, and future approaches has also been featured.

Bibliography

[1] Michael J. Murphy et al. "Experiences Building and Deploying Wireless Sensor Nodes for the Arctic Tundra." In: *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. 2021, pp. 376–385. DOI: 10.1109/CCGrid51090.2021.00047.

[2] *Norway (except Svalbard)*. https://eu-interact.org/norway-except-svalbard/. (Accessed on 2024-05-28).

[3] *Svalbard, Norway*. https://eu-interact.org/svalbard-norway/. (Accessed on 2024-05-28).

[4] *Distributed Arctic Observatory*. https://en.uit.no/project/dao. (Accessed on 2024-04-09).

[5] *Climate-ecological Observatory for Arctic Tundra*. https://coat.no/. (Accessed on 2024-04-09).

[6] W.R. Heinzelman, A. Chandrakasan, and H. Balakrishnan. "Energy-efficient communication protocol for wireless microsensor networks." In: *Proceedings of the 33rd Annual Hawaii International Conference on System Sciences*. 2000, 10 pp. vol.2-. DOI: 10.1109/HICSS.2000.926982.

[7] *ns-3 Network Simulator*. https://www.nsnam.org/. (Accessed on 2024-05-06).

[8] *OMNeT++ Discrete Event Simulator*. https://omnetpp.org/. (Accessed on 2024-05-06).

[9] *Simpy Discrete Event Simulation for Python*. https://simpy.readthedocs.io/en/latest/index.html. (Accessed on 2024-05-06).

[10] Jennifer Yick, Biswanath Mukherjee, and Dipak Ghosal. "Wireless sensor network survey." In: *Computer Networks* 52.12 (2008), pp. 2292–2330. ISSN: 1389-1286. DOI: https://doi.org/10.1016/j.comnet.2008.04.002. URL: https://www.sciencedirect.com/science/article/pii/S1389128608001254.

[11] Douglas S. J. De Couto et al. "A high-throughput path metric for multi-hop wireless routing." In: *Proceedings of the 9th Annual International Conference on Mobile Computing and Networking*. MobiCom '03. San Diego, CA, USA: Association for Computing Machinery, 2003, pp. 134–146. ISBN: 1581137532. DOI: 10.1145/938985.939000. URL: https://doi-org.mime.uit.no/10.1145/938985.939000.

[12] Ian F. Akyildiz, Xudong Wang, and Weilin Wang. "Wireless mesh networks: a survey." In: *Computer Networks* 47.4 (2005), pp. 445–487. ISSN: 1389-1286. DOI: https://doi.org/10.1016/j.comnet.2004.12.001. URL: https://www.sciencedirect.com/science/article/pii/S1389128604003457.

[13] Ted Faison. *Event-Based Programming*. Springer, 2006.

[14] Sasu Tarkoma. "Publish/subscribe systems: design and principles." In: John Wiley & Sons, 2012. Chap. 1.

[15] Frank Dabek et al. "Event-driven programming for robust software." In: *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*. EW 10. Saint-Emilion, France: Association for Computing Machinery, 2002, pp. 186–189. ISBN: 9781450378062. DOI: 10.1145/1133373.1133410. URL: https://doi-org.mime.uit.no/10.1145/1133373.1133410.

[16] *Async IO in Python: A Complete Walkthrough*. (Accessed on 2024-05-27).

[17] Wafa Akkari, Badia Bouhdid, and Abdelfettah Belghith. "LEATCH: Low Energy Adaptive Tier Clustering Hierarchy." In: *Procedia Computer Science* 52 (2015). The 6th International Conference on Ambient Systems, Networks and Technologies (ANT-2015), the 5th International Conference on Sustainable Energy Information Technology (SEIT-2015), pp. 365–372. ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2015.05.110. URL: https://www.sciencedirect.com/science/article/pii/S1877050915009102.

[18] Mudathir F. S. Yagoub et al. "Lightweight and Efficient Dynamic Cluster Head Election Routing Protocol for Wireless Sensor Networks." In: *Sensors* 21.15 (2021). ISSN: 1424-8220. DOI: 10.3390/s21155206. URL: https://www.mdpi.com/1424-8220/21/15/5206.

[19] S. Lindsey and C.S. Raghavendra. "PEGASIS: Power-efficient gathering in sensor information systems." In: *Proceedings, IEEE Aerospace Conference*. Vol. 3. 2002, pp. 3–3. DOI: 10.1109/AERO.2002.1035242.

[20] Arati Manjeshwar and Dharma P Agrawal. "TEEN: ARouting Protocol for Enhanced Efficiency in Wireless Sensor Networks." In: *ipdps*. Vol. 1. 2001. 2001, p. 189.

[21] Sigurd Karlstad. "Clock Synchronization between Observational Units in the Arctic Tundra." MA thesis. UiT Norges arktiske universitet, 2021.

[22] Loic Guegan and Issam Raïs. "Simulation of Distributed Systems in Constrained Environments Using ESDS: the Arctic Tundra Case." In: *2023 IEEE International Conferences on Internet of Things (iThings) and IEEE Green Computing & Communications (GreenCom) and IEEE Cyber, Physical & Social Computing (CPSCom) and IEEE Smart Data (SmartData) and IEEE Congress on Cybermatics (Cybermatics)*. 2023, pp. 547–554. DOI: 10.1109/iThings-GreenCom-CPSCom-SmartData-Cybermatics60724.2023.00104.

[23] Drew Gislason. *Zigbee wireless networking*. Newnes, 2008.